TexBench: Harnessing LLMs for Efficient Key-Value Benchmarking

Shubham Kaushik[1⋆], Abhishek Chanda[2⋆], and Subhadeep Sarkar[1]

[1] Brandeis University, MA, USA
{kaushiks,subhadeep}@brandeis.edu
[2] Independent Researcher
abhishek.becs@gmail.com

Abstract. Key-value stores are widely adopted to serve modern heterogeneous workloads, yet selecting the best key-value store for a given workload and target performance remains an incredibly complex and time-consuming task, even for expert DBAs. In practice, this entails extensive, large-scale benchmarking and analysis involving dozens of candidate key-value stores. Comparing databases using existing key-value benchmarks, such as YCSB, KVBench, db_bench, and Tectonic, has several challenges. (A) It requires a substantial effort in setting up the benchmarking environment and connecting the databases, along with a thorough understanding of benchmark-specific configurations and tuning parameters. Further, existing benchmarks either (B) are completely unable to capture the dynamic and complex nature of real-world workloads, or (C) rely on fixed, templated benchmark workloads, with limited customization options. Thus, benchmarking key-value stores against application-specific, production-like workloads is difficult and does not scale, leading to suboptimal performance.

We introduce *TexBench*, a *unified key-value benchmarking suite* that abstracts the complexities of workload configuration, benchmark setup, and database connections behind a unified interface, enabling users to evaluate multiple key-value stores against production-like workloads, and visually compare their performance side-by-side. To emulate the dynamic and complex properties of real-world workloads, *TexBench* uses Tectonic – a Rust-based, highly configurable key-value benchmark – under the hood. To facilitate benchmarking against custom workloads, *TexBench* integrates a large language model (LLM) plugin that captures benchmarking specifications expressed in natural language and translates them into a JSON format interpretable by Tectonic. *TexBench* currently supports four key-value stores – ScyllaDB, RocksDB, Cassandra, and Redis, and performs up to 16.7× faster in terms of workload generation compared to raw LLM prompting. The source code of *TexBench* is available at https://github.com/SSD-Brandeis/TexBench.

Keywords: Key-value benchmark · Shifting workloads · LLM.

⋆ Authors with equal contribution.

1 Introduction

Key-Value Stores and their Rich Design Space. Key-value stores are widely adopted across industry as they offer high ingestion throughput, efficient storage of heterogeneous data, and a rich and tunable design space [19,24,38,40]. Examples of production deployments of key-value stores include RocksDB [15] at Meta, Rockset [37] at OpenAI, BigTable [9] and LevelDB [17] at Google, Cassandra [2], and HBase [3] from Apache, Redis Cache at Redis [36], and FASTER [8] and F2 [23] at Microsoft. However, the performance of these storage engines varies by several orders of magnitude if there is a shift in the workload characteristics [18,26,42,39], a change in the underlying hardware [10,11,22,29], or an update to the performance goals [5,13,20]. Thus, identifying a suitable storage engine and its optimal tuning for a given workload and target performance is an extremely complex and tedious endeavor that entails extensive large-scale benchmarking and analysis of the benchmarking results [25,28,42,50].

State-of-the-art Key-Value Benchmarks and their Limitations. Existing key-value benchmarking suites, such as YCSB [12], db_bench [14], and KVBench [51] offer a fixed set of benchmarks. For instance, YCSB outlines six benchmark workloads (YCSB-A through YCSB-F) to model common access patterns for key-value workloads. However, these benchmarks suffer from several key limitations that render benchmarking against production-like workloads difficult, if not impossible. For instance, YCSB is unable to benchmark databases against delete-heavy workloads, which is common in production [7,41,52]. It is also unable to support operation-specific distributions among several other limitations, as listed in Table 1. db_bench is tightly coupled to RocksDB and, therefore, is virtually unusable for benchmarking other key-value stores and comparing database performance [14]. KVBench is effectively a workload generator that can generate a wide variety of workloads, but lacks the infrastructure of a benchmarking suite [51]. Furthermore, none of these benchmarks support benchmarking dynamically shifting workloads in which properties evolve over time, despite it being a common phenomenon in key-value applications. For instance, the query frequency in UDB, a social graph workload at Meta, shoots up by $\sim 10\times$ on weekends for certain column families [7].

In our prior work, Tectonic [33], we built the first key-value benchmarking suite that supports emulation of dynamically shifting, realistic workloads. In addition, Tectonic can benchmark databases against all YCSB, db_bench, and KVBench benchmark workloads out of the box, as it is founded on a highly configurable and rich specification design space. This flexible design and backward compatibility, however, come at the cost of composing a complex and comprehensive JSON specification input file. Tectonic uses this JSON as input to model the benchmarking workload, which, in effect, requires users to be familiar with Tectonic's design specification when initiating benchmarking.

Challenge 1: Expressing Realistic Workloads. Modeling a production workload in a benchmarking suite requires understanding how to map real-world workload operations into the benchmark configuration design space. For in-

Table 1. *TexBench* is the only suite that supports text-to-workload generation, cross-benchmark evaluation, and side-by-side database comparison with a unified interface.

		YCSB	KVBench	db_bench	Tectonic	*TexBench*
operations	insert	●	●	●	●	●
	update	●	●	●	●	●
	read-modify-write	●	–	●	●	●
	point query	●	●	●	●	●
	empty point query	–	●	●	●	●
	range query	●	●	●	●	●
	point delete	–	●	●	●	●
	empty point delete	–	●	–	●	●
	range delete	–	●	●	●	●
distributions	uniform	●	●	●	●	●
	normal	–	●	●	●	●
	beta	–	●	–	●	●
	zipfian	●	●	–	●	●
	exponential	–	–	●	●	●
	log normal	–	–	–	●	●
	poisson	–	–	–	●	●
	weibull	–	–	–	●	●
	pareto	–	–	●	●	●
properties	dynamic workload shifts	–	○	○	●	●
	context-aware shifting	–	–	–	●	●
	data sortedness	–	–	–	●	●
	variable query selectivity	●	–	●	●	●
	variable key-value length	–	–	–	●	●
	temporality-based access	–	–	●	●	●
	customizable key prefix	–	–	○	●	●
	composite keys	–	–	–	●	●
	benchmarking databases	●	–	○	–	●
	unified framework	–	–	–	–	●
	visualized comparison	–	–	–	–	●
	text-to-workload (LLM)	–	–	–	–	●
	support preset workloads	–	–	–	–	●

● supported ○ requires significant manual effort – not supported

stance, to model shifting workloads, one needs (i) to identify the different phases of the workload, and then (ii) for each phase, model the composition, distribution, order, and key-value properties of the constituent operations [7,33]. While Tectonic allows a user to model shifting workload properties, it requires a thorough understanding of Tectonic's internal constructs and syntax, which slows down the process of benchmarking.

Challenge 2: Lack of a Unified Benchmarking Framework. The complexity of benchmarking compounds as new databases, each with their own unique APIs and configurations, are added to the benchmarking suite. The databases are often written in different programming languages and require different client libraries and connection protocols, which makes their integration into the benchmarking suite difficult. Further, comparing databases against the same workload entails separately running the benchmark for each database and manually comparing the benchmarking results. If the specific workload is not already captured by the existing benchmarks, users need to settle for the closest match or spend a considerable amount of effort to construct one by hand. Clearly, this process does

not scale as it is incredibly complex, time-consuming, and error-prone, especially when dealing with multiple databases.

Solution: *TexBench*. To address this, we introduce *TexBench*, an *accessible, unified, LLM-driven key-value benchmarking suite that enables benchmarking databases against dynamically shifting and complex production-like workloads and comparing database performance side by side*. *TexBench* builds on our prior work Tectonic, a Rust-based, highly configurable, scalable, and resource-efficient workload generator that is curated to emulate *multi-phased shifting workloads*, along with support for a *rich set of complex workload features*, including *variable data sortedness* and *custom data formats* [33]. To facilitate seamless workload customization without requiring knowledge about the underlying benchmarking suite, we integrate *TexBench* with a large language model (LLM) plugin that allows users to specify workloads in natural language and execute benchmarks across selected key-value stores. Further, *TexBench* interactive user interface (UI) allows users to benchmark and compare the performance of multiple databases against the same workload visually and side by side.

Contributions. The overarching goal of *TexBench* is to make database benchmarking more accessible, straightforward, and insightful for the broader database community, including experts, researchers, and inexperienced yet curious users. The contributions of this work are as follows.

(*1*) *Benchmark Against Existing Presets:* We designed *TexBench* to enable cross-benchmark benchmarking of databases, which includes YCSB, KVBench, db_bench, and Tectonic. Users choose from a list of available standard benchmarks, set the scale as required, and *TexBench* takes care of the rest.

(*2*) *Generate Complex User Described Workloads:* We built *TexBench* with an LLM-plugin that allows users to describe their own customized benchmarks in natural language to meet application requirements. An LLM translates the user-defined requirements into workload configuration, which Tectonic then uses to generate and execute workloads on the selected key-value stores.

(*3*) *A Unified Framework: TexBench* offers a simplified framework for benchmarking and comparing databases side by side, all in one place. This makes it easy, even for beginners, to describe workload and benchmark multiple databases to find a suitable one for their application.

TexBench abstracts all the complexities behind a unified UI, simplifying the process of database benchmarking. Users can also customize, verify, modify, and download the workload and benchmarking log from the same interface.

2 *TexBench*: A Unified Benchmarking Suite

TexBench is a unified, user-first key-value benchmarking framework that leverages Tectonic's highly configurable rich design space (see Table 1) and the natural language processing capabilities of LLMs [28,49]. It is encapsulated by an

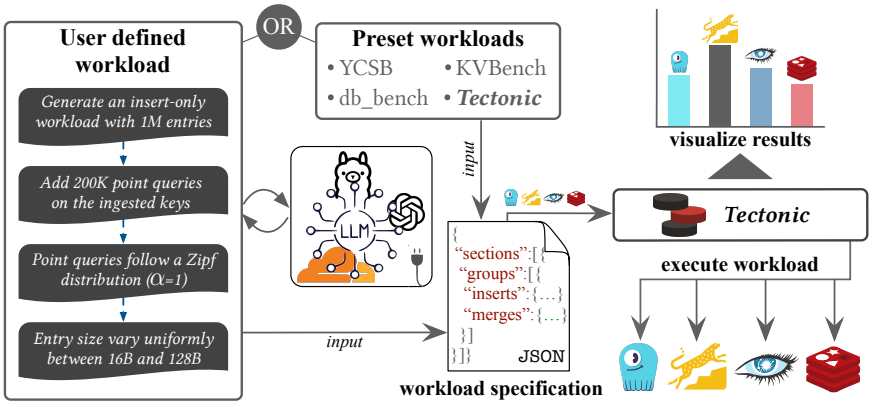


Fig. 1. *TexBench*'s design to generate a workload and benchmark multiple databases.

intuitive UI that allows users to benchmark multiple databases against the same workload and compare their performance visually by simply specifying their candidate databases and the benchmarking workload. *TexBench* aims to speed up the benchmarking setup process and reduce the amount of prior knowledge required before running the first benchmark. To the best of our knowledge, *TexBench* is the first key-value benchmarking suite that supports (i) benchmarking databases under dynamic workloads and (ii) cross-benchmark evaluation of databases, under a unified framework that allows for (iii) workload specification in natural language and (iv) side-by-side performance comparison of multiple databases through an intuitive UI.

2.1 *TexBench* Architecture: An Overview

TexBench, at a high level, has four key components: (i) a UI-based input prompt, (ii) Tectonic integration, (iii) an LLM-plugin, and (iv) an output UI to render results, see Fig. 1. Users can run the standard key-value benchmarks out of the box using the input prompt, without installing and setting up the various benchmarking suites individually. For instance, evaluating a database against YCSB-A [12] and KVBench-II [51] does not require installing both YCSB and KVBench – they can be simply initiated by choosing the appropriate benchmark family and workload. The LLM-plugin allows users to generate custom workloads by simply presenting the workload specifications in natural language.

The *TexBench* Execution Pipeline. Benchmarking in *TexBench* happens over three steps, as shown in Fig. 1.

Step 1: Specifying the Benchmarking Workload. *TexBench*'s input UI allows users to construct a real-world workload configuration in two ways, as shown in Fig. 1. (A) Users can choose from a list of existing benchmark families – YCSB, KVBench, db_bench, and Tectonic – where the specifications are readily available in accordance with the Tectonic input standard, and can be scaled through *TexBench* as needed. *TexBench* supports all six YCSB benchmark workloads (YCSB-A through YCSB-F), five KVBench benchmarks (KVBench-I through

KVBench-V), seven db_bench benchmarks, and seven Tectonic benchmarks (Tectonic-I through Tectonic-VII), with user-specified scale factors. (B) Alternatively, the users can construct custom workloads by either (i) using a simplified workload editor or (ii) interactively by prompting the LLM. This option allows users to generate dynamic workloads where the workload properties, such as operation distribution, frequency, size of the key-value pairs, etc., vary with time across different phases of a workload. Additionally, (C) *TexBench* allows users to start with an existing benchmark, such as YCSB-B (read-heavy), and tailor it to their specific needs, such as adding support for deletes, thereby generating realistic workloads that closely match the use case at hand. *TexBench* parses the LLM response through a specialized domain-specific language (refer to §2.3) to ensure the correctness of workload specification. Further, users can choose the number of threads to run when generating a synthetic workload and benchmarking databases to simulate the realistic environment.

Step 2: Choosing the Candidate Databases. Once the target workload is ready, users need to select the databases to benchmark against. For this, users mark checkboxes in the same UI from the available list. Currently, *TexBench* supports evaluating four databases (i) RocksDB [15], (ii) Cassandra [2], (iii) ScyllaDB [43], and (iv) Redis [36], with several more in development. A user may select multiple databases to benchmark against the same workload. *TexBench* will benchmark the databases sequentially and render the results side-by-side for comparison.

Step 3: Benchmarking and Comparing Databases. *TexBench* spawns a Tectonic instance for each selected key-value store to avoid noise in the benchmark results. When the benchmarking for one database finishes, *TexBench* parses the results, updates the UI, and runs the next database. This setup is more transparent because the user does not have to worry about hardware, installing multiple tools, or manually comparing results, since everything is handled in a unified UI.

Infrastructure and Integrated Models. *TexBench* runs as a self-contained benchmarking suite, where the UI is designed using a lightweight `Node.js` (version 18+) JavaScript runtime. Tectonic is compiled as a single binary, allowing users to install *TexBench* along with its dependencies. *TexBench* currently supports three LLM backends: Ollama (Llama 3, Llama 3.1, Llama 3.2) [45] for fully local inference, OpenAI (GPT-4o, GPT-4-turbo) [32], and Cloudflare. Users can switch between backends/models either at startup by setting a single environment variable (`AI_PROVIDER`) or through the UI. Each database is integrated through a unified adapter layer that normalizes operations and benchmark results in a common format. Adding a new database or LLM-backend only requires integrating the same interface.

2.2 Why Tectonic as an Execution Engine?

Tectonic is a highly configurable, resource-aware key-value benchmark that supports a wide range of operations, operations-specific distributions, and complex characteristics, such as dynamically shifting patterns, varied data sortedness, and custom data formats, ensuring it closely mimics the real-world work-

loads [33]. Tectonic enables *TexBench* to efficiently model realistic, customized, user-defined modern application workloads to precisely benchmark databases.

```
{
 "sections": [ {                                                    <SECTION 1>
  "groups": [ {                                                      <GROUP 1>
   "point_queries": { "op_count": 500000,
                      "selection": { "beta": { "alpha": 0.1, "beta": 5 }}},
   "updates": { "op_count": 500000,
                "val": { "uniform": { "len": 128 }},
                "selection": { "beta": { "alpha": 0.1, "beta": 5 }}}
  }, { ... <GROUP 2> } ]}, {... <SECTION 2> } ]
}
```

Fig. 2. An example JSON specification that Tectonic uses as input.

Tectonic takes a JSON specification file as an input describing *phase-wise* workload specifications using *Sections* and *Groups* (phases). A *Section* consists of a list of *Groups* that share the same key domain, where each *Group* is a set of interleaved operations, such as inserts, empty and non-empty point queries, range queries, updates, empty and non-empty deletes, and read-modify-writes. Phases that operate on disjoint key domains are put into separate *Sections*. By default, Tectonic generates each *Section* and *Group* sequentially, but supports parallelism whenever applicable and configured by the user. Fig. 2 shows an example specification of what Tectonic expects as an input.

Challenges for LLM Integration. LLMs are trained on vast amounts of data, enabling them to effectively analyze and perform well-defined tasks easily [32,45]. However, there are certain challenges when LLMs are used for specific translation tasks, for example, translating user-described prompts into a workload (JSON) specification while respecting the benchmark-defined schema. (i) The recent advancements in LLMs have significantly increased the context window sizes, but performance degrades when relevant information is buried in the middle of a long context, as studied in *Lost-in-the-Middle Effect* [31]. (ii) LLMs are prone to *hallucinating unintended text* in every response, making it challenging to extract the useful information from the bubble of text and to trust the accuracy [21]. (iii) More often, LLMs respond with *unexpected and invalid results* (might be because of ambiguous human language), which can break the entire benchmarking pipeline due to incorrect workload specification [4,6]. (iv) Lastly, *processing LLM response* and making desired changes while ensuring the correctness of the configuration file expected by Tectonic is non-trivial. *TexBench*'s LLM-plugin addresses these by using a customized domain-specific language, enabling seamless pipeline execution.

2.3 *TexBench*'s LLM-Plugin

The key intuition behind integrating LLMs into *TexBench* is *not to let LLMs generate a JSON specification directly*. Instead, the LLM produces a sequence of commands using domain-specific language (DSL) and a described grammar in the prompt. These commands are used to mutate the JSON and fulfill the

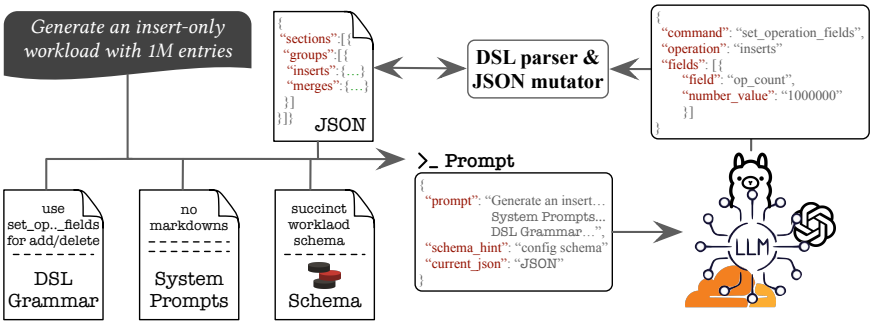


Fig. 3. Workflow of the *TexBench* LLM-plugin translating natural language user prompts into DSL commands.

user's custom workload requirements. The LLM-plugin consists of: (i) a domain-specific language which sets boundaries on LLM response, (ii) 44 rules, including a fixed set of `System Prompts` and DSL grammar, to teach LLMs the scope of workload specification design space, and (iii) a parser that extracts commands from LLM response and applies them onto the target JSON, see Fig. 3.

Domain-Specific Language. The DSL is a constraint language that defines how LLMs should respond to the users' prompts such that *TexBench*'s deterministic parser can perform semantic validation for the suggested change while mutating JSON. DSL ensures the safety and correctness of the benchmarking pipeline by keeping the JSON specification always valid. To this end, DSL defines the following eight commands, mapping user prompts to DSL commands:

1. **`set_top_field`.** This command sets the base property of a `Section` or `Group`. For instance, to constrain key generation to alphanumeric characters, this command suggests setting the `character_set` property to `alphanumeric`.

2. **`clear_operations`.** While editing an existing benchmark, a user might want to remove a few operations; this command helps clear existing operations from the workload specification. For example, removing updates from the execution phase of the YCSB-B (read-heavy) workload, making it 100% reads.

3. **`set_operation_fields`.** A frequently used command that helps add a new operation to the workload specification or modifies the properties of an existing operation, such as frequency (*op_count*), distribution, and range. For example, Fig. 4 shows an LLM response for a user prompt requesting to add 50K point query operations to the workload.

4. **`scale_all_op_counts`.** To benchmark the databases with a scaled workload, users can ask LLMs to scale up or down the existing workload configuration. This factor should be a positive floating-point number and can also be used to scale standard benchmarks such as YCSB, KVBench, db_bench, and Tectonic.

5. **`set_range_scan_length`.** This command sets the scan length of an operation to a deterministic selectivity value. For instance, if `scan_length` is 100, then `selectivity` is computed locally in *TexBench* as $100/N$, where N is the

```
{
  "summary": "Enable 50k point queries",
  "commands": [{
      "kind": "set_operation_fields",
      "operation": "point_queries",
      "fields": [
        { "field": "op_count", "number_value": 50000 },
        { "field": "enabled", "boolean_value": true }
      ]}],
  "clarifications": [...], "assumptions": [...]
}
```

Fig. 4. LLM response suggesting "`set_operation_fields`" to set the count of `point_queries` to 50,000 based on user request.

total number of inserts. Fig. 5 shows an LLM response suggesting to update `scan_length` of `range_queries` operation, in `Group 2` of `Section 1`, to 100.

```
{
  "kind": "set_range_scan_length",
  "operation": "range_queries",
  "section_index": 1,
  "group_index": 2,
  "scan_length": 100
}
```

Fig. 5. LLM response suggesting "`set_range_scan_length`" to reset `scan_length` to 100 for `range_queries` in `Group 2` of `Section 1`.

6. **`append_group`.** Modern application workloads typically consist of multiple phases, and a user constructing a custom workload requires adding multiple phases. The `append_group` command appends a new `Group` to a `Section`.

7. **`rename_group_operation`.** To rename an operation, such as changing insert to update, DSL defines a `rename_group_operation` command that preserves the properties of an existing operation and converts it to another operation. It might not be possible to convert point queries to range queries, as range queries require additional parameters, such as scan length. DSL parser catches and resolves these errors by replacing them with defaults.

8. **`replace_sections`.** This command replaces the entire `Section`/`Group` with a new one. It is useful when tailoring a multiphase workload configuration. For example, a user can add a read-heavy phase after a write-heavy phase by appending a new point-query-heavy `Group`.

Domain-Specific Grammar. The grammar that defines exactly which DSL command LLMs must emit for a given user intent. Fig. 6 shows a few examples of *user intent* $\rightarrow$ *DSL* commands and Tectonic vocabulary, which is fed into the LLM prompt. Full documentation covering all 30 grammar rules is available on GitHub[3]. Consider a user asking to add a new group to the configuration; the LLM will respond with the `append_group` command, whereas if a user asks to update an existing operation count, the LLM responds with the

[3] *TexBench*'s source code: https://github.com/SSD-Brandeis/TexBench.

— ***DSL grammar (user intent → DSL command)*** —

targeted-edit (add/remove/edit) → `set_operation_fields`
bulk-scale (shrink/grow/all operations) → `scale_all_op_counts`
scan-edit (scan length/selectivity) → `set_range_scan_length`
phase-append (add/new/phase/group) → `append_group`
layout-rewrite (rewrite/replace/construct) → `replace_sections`

— ***Tectonic vocabulary*** —

operation ::= `inserts` | `updates` | `merges` | `point_queries` | `range_queries` | `point_deletes` | `range_deletes` | `empty_point_queries` | `empty_point_deletes` | `sorted`
field_entry ::= ⟨ *name*, *value_slot* ⟩
value_slot ::= `string_value` | `number_value` | `boolean_value` | `json_value`

Fig. 6. *TexBench*'s DSL grammar, mapping natural-language intent to DSL.

— ***output structure*** —

response ::= ⟨ `summary`, `command`, `clarifications`, `assumptions` ⟩
format ::= JSON *(no markdown, no extra prose)*
content ::= *workload-spec* *(not sample data)*

— ***defaults & normalization*** —

unspecified → *safe-default* *(picks default value)*
numeral ::= *n*`KB` → *n*×1024 | *n*`K` → *n*×1,000 | *n*`M` → *n*×1,000,000

Fig. 7. *TexBench*'s `System Prompts` to bound LLM response.

`set_operation_fields` command. Human language can be ambiguous; therefore, LLMs must follow standard rules defined by a DSL grammar to respond appropriately and ensure the JSON specification remains valid.

Hard coded `System Prompts`. Beyond DSL grammar, *TexBench* injects 14 hard-coded `System Prompts` with every request, forcing LLMs to respond within the valid configuration space, see Fig. 7. For instance, if the user asks for the key-value size to be 1KB, the LLM must respond in bytes.

***TexBench*'s Curated LLM Prompt.** The message prompt that is sent to the LLM for every user request consists of (i) the user-entered message, (ii) DSL grammar and `System Prompts`, (iii) a succinct Tectonic schema describing supported operations and their allowed properties, and (iv) current workload JSON. This allows LLMs to understand the scope of the overall design space, the current specification, and the boundaries to respect when suggesting a change using DSL. An example prompt is shown in Fig. 3.

DSL Parser & Mutator. The *TexBench*'s parser extracts the sequence of commands responded by the LLM and passes them through a rule-based check against the schema provided by Tectonic. If everything matches, the parser applies the requested change to the JSON, updating the workload specification.

Table 2. *TexBench* produces specification[5] close to user intent; raw prompting collapses on a local model and *TexBench* improved accuracy (intent match) on GPT-4o.

	Llama 3.2		**GPT-4o**	
category	**raw prompt**	***TexBench***	**raw prompt**	***TexBench***
simple	4.8	**79.2**	95.2	**100**
mixed	5.8	**65.4**	91.9	**96**
phased	4.1	**56.3**	88.1	**93.7**
complex	3.9	**44.4**	90.2	**95.7**
overall	4.7	**61.3**	91.3	**96.3**

However, if the command is invalid, the parser ignores that specific change to keep the specification valid.

3 Experimental Evaluation

In this section, we first verify the correctness of the LLM-generated input workload configuration file that Tectonic uses to emulate realistic workloads. For this, we used 50 user prompts from four workload categories, as presented below.

1. *Simple:* A set of 12 prompts where the task is straightforward.
   Example: "*Generate an insert-only workload with 100,000 operations.*"
2. *Mixed:* A set of 13 prompts where multiple operations need to be generated.
   Example: "*Generate a workload with 800,000 inserts and 200,000 point queries (80% inserts, 20% reads), totaling 1,000,000 operations.*"
3. *Phased:* A set of 13 prompts to generate workloads consist of multi-phases.
   Example: "*Create a two-phase workload: first insert 1,000,000 keys (phase 1), then run 500,000 point queries on those keys (phase 2). The two phases must be separate groups so that phase 2 reads keys written in phase 1.*"
4. *Complex:* 12 production-like workload prompts that include changing several operation-specific properties such as distribution, op_count, and selectivity.
   Example: "*Create a production-like two-phase workload: phase 1 inserts 2,000,000 records with 64-byte keys and 256-byte values; phase 2 runs 1,800,000 point queries with a Zipf selection distribution (n=2,000,000, s=0.99) and 200,000 range queries with 1% selectivity, all interleaved.*"

Next, we show that benchmarking and analyzing performance against standard benchmarks is insufficient, as real-world workloads differ and shift dynamically over time, thereby changing the choice of a suitable database.

Experimental Setup. We use a machine with two Intel Xeon Platinum 8380 CPUs (2.30 GHz), 80 cores with 160 threads, an NVIDIA A100 PCIe GPU

[5] *TexBench* always emits a schema-valid specification by construction; its *intent match* value measures the extent to which the user request was properly fulfilled. Raw prompting may even fail to produce a schema-valid specification (e.g., Llama 3.2 returns garbage values), so its values reflect both schema validity and intent.

Table 3. Case study on *Complex* workloads: Raw prompting (GPT-4o) flattens the two phases and gives the wrong workload specification; *TexBench* produces it correctly.

dimension/score	**maximum**	**raw prompt**	***TexBench***
syntactic validity	5	5	5
section cardinality	5	5	5
phase cardinality	10	5	10
key-domain	5	5	5
phase ordering	10	0.0	10
operation composition	20	8	20
operation properties	35	18.7	35
schema correctness	10	10	10
total	100	**56.7**	**100**

(40GB), 256GB of DDR4 RAM, 120MB L3 cache, and a 480GB SSD, running Ubuntu 24.04.4 LTS [27]. A100 GPU is used to accelerate inference for Ollama.

Setup 1: We experiment with two models: Llama 3.2 from Ollama (free) and GPT-4o from OpenAI (paid). Using a list of 50 prompts, we send two types of requests to each model: raw prompts and *TexBench*-prepared prompts. The raw prompt is prepared by appending the Tectonic workload schema to the user-described requirement to get a full specification JSON file. *TexBench*'s prepared prompt includes `System Prompts`, DSL grammar rules, and succinct version of Tectonic's configuration schema. For the analysis, we compare raw prompting with *TexBench* across eight dimensions, ranging from syntactic validity to schema correctness, and assess how well LLMs capture user intent and generate the expected configurations and DSL commands. We also compare the round-trip time (RTT) and cost to generate a configuration for the paid model. Timeouts are set to 120s and 200s for *TexBench* and raw prompting, respectively.

***TexBench* Preserves the Correctness of Specification.** *TexBench* response produces a valid JSON specification file, fulfilling user intent for a custom workload. Table 2 shows that raw prompting is not scalable, especially for the local free models, as the workload schema files are lengthy, leading to slowdown, the "lost-in-the-middle" effect [31], and hallucinated responses. Llama 3.2 model with raw prompting captures user intent only 4.7% of the time across 50 prompts, failing in each category. On the other hand, *TexBench*, using the same model, captured the user requirements in 61% of cases. As the prompts become more complex, all models with raw prompts, as well as *TexBench*, show a drop in accuracy. However, *TexBench* always wins over raw prompting even for paid models like GPT-4o, showing 96.3% accuracy compared to raw prompting, i.e. 91.3%. The responses are further parsed by the DSL parser for verification. Table 3 shows a complex category use case with a breakdown of scoring that we use to model this experiment. If a model response returns a proper JSON, the correct number of sections, and a key domain, our model gives 5 credits per category. Further, it rewards 10 points for the correct number of phases and 10 for the correct phase ordering, such as write-heavy before read-heavy, if a user asks

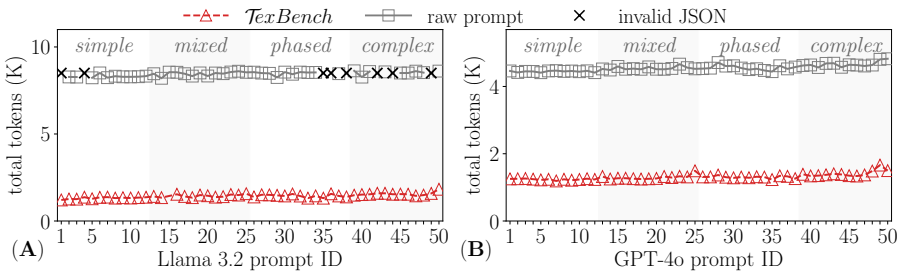


Fig. 8. *TexBench* reduces the prompt size, regardless of the LLM, by structuring required information succinctly and thereby reducing monetary costs.

for phase 1 to be write-heavy and phase 2 to be read-heavy. Operation-specific properties are complex and can vary per operation. For example, inserts cannot have selectivity, whereas range queries must. Similarly, an insert must have key and value properties, and point queries should only have key. We model it with 35 points and implement a partial-points mechanism using minimum-over-maximum ratio scoring when a value is close to what the user asked for. For example, if the user requests 1M insert operations and the LLM returns 0.9M, we award partial credit (90% of the category). Lastly, if the specification is fully correct, models are awarded 10 bonus points. For a complex use case, raw prompting failed to generate the proper phases, resulting in a loss of 10 points and a few more for setting incorrect values to operation properties.

***TexBench* Reduces API Cost.** Enterprise LLMs charge clients on a per-token basis; therefore, the size of each request determines the total cost. Raw prompting embeds the entire Tectonic schema (≈ 4,300 input tokens) in every request, leaving no room for information such as hard-coded rules to guide LLMs in preparing a structured response. Llama 3.2 responses bloat, leading to nearly 2× more token usage than GPT-4o; see Fig. 8. *TexBench* sends a succinct schema hint and `System Prompts` (≈ 1,100 input tokens), leaving headroom to teach the LLM how to structure its output, with 3.5× fewer tokens than raw prompting, see Fig. 8. For GPT-4o, over the 50 prompts, raw prompting costs $0.64 in total, whereas *TexBench* costs only $0.23 due to its small yet effective prompt strategy. *TexBench* reduces the API cost of using LLMs to generate custom workloads by 2.8× compared to prompting blindly.

DSL Improves RTT. In Fig. 9(A) and 9(B), we show that *TexBench*-specified system prompts and DSL grammar help in achieving better RTT when using the LLM-plugin and also ensure the correctness of the response and JSON specification. On Llama 3.2, *TexBench* responses are, on average, 16.7× faster than raw prompting – raw prompting took more than 150*s* for 39 out of 50 requests, and 8 of those exceeded the context limit, resulting in a null response. On GPT-4o, raw prompting returns valid JSON, but *TexBench* outperforms, achieving responses 1.2× faster.

Setup 2: In this experiment, we show how state-of-the-art key-value benchmarks, such as YCSB, fall short in emulating shifting workloads. We use two workloads: (i) a static YCSB-A workload and (ii) an LLM-assisted shifting work-

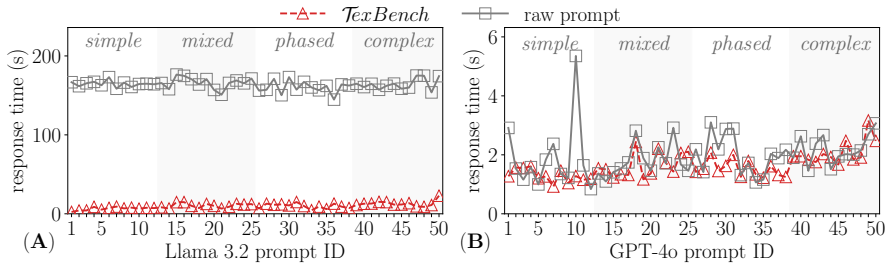


Fig. 9. Concise prompts and clear requirements improve *TexBench* DSL round-trip-time for (A) Llama 3.2 while avoiding timeouts and (B) improving GPT-4o time.

load, to evaluate four databases, ScyllaDB, RocksDB, Cassandra, and Redis. The 3-phase dynamic workload shifts from write-heavy to read-heavy over time. The total number of operations in each phase is 2M. Phase 1 has 100% unique inserts, followed by 20% updates interleaved with 80% point queries in Phase 2, before having 100% point queries in Phase 3. The key and value sizes are increased from 16B to 24B and from 64B to 1024B, respectively.

***TexBench* Emulates Dynamic Workloads Significantly More Accurately than YCSB.** Fig. 10 shows the performance shifts across four databases as the workload shifts from YCSB-A to a shifting workload. On YCSB-A, RocksDB offers a throughput of 210Kops/s, outperforming Redis (27Kops/s), ScyllaDB (3.3Kops/s), and Cassandra (1.7Kops/s) by up to 123×. However, under a shifting workload, none of the databases performed similarly to how they performed on YCSB-A: RocksDB and Redis throughput dropped the most, retaining only 75% and 76% of their YCSB-A throughput, as shown in Fig. 10(A). Cassandra and ScyllaDB show stable performance, sustaining 85% and 92% of their throughput, respectively. The operation-level breakdown in Fig.10(B) shows that this drop in performance is operation- and engine-specific. Redis's average inserts slowed by 114%, and RocksDB's insert and update slowed by 60% and 39%, respectively, whereas point query latency actually improved by 30%, mostly due to block cache exploiting the skewed access pattern. Cassandra and ScyllaDB exhibit smaller but similar performance variation. Benchmarking against standard benchmarks like YCSB-A is insufficient as performance varies significantly across workloads, which reinstates the importance of *TexBench* that is able to capture the temporal shifts in workload characteristics.

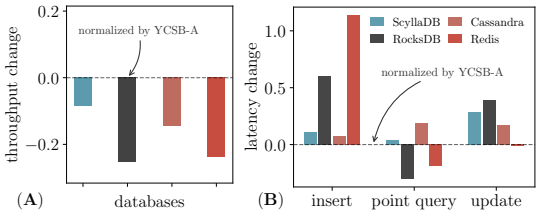


Fig. 10. Performance varies by several orders of magnitude when databases are deployed in production due to uncaptured properties of realistic workloads by the standard benchmark YCSB-A.

Discussion: Saves Benchmark Setup Time. Benchmarking a list of candidate databases with state-of-the-art benchmarking suites is a multi-step, repet-

itive, and error-prone process. Evaluating multiple databases against a multi-phased workload using YCSB requires: (i) capturing workload properties according to YCSB's configuration syntax while mapping each phase to its closest match, (ii) modifying source code to output workload into a file, (iii) stitching all phases to a single file, (iv) modifying YCSB to read and execute from a file, (v) launching separate benchmarks for each database to record the results, and then, (vi) normalizing and manually comparing their performance by parsing the individual output files. The stitched workload is equivalent to a random workload, as two separate phases are generated through distinct key domains; thus, this entire pipeline provides inaccurate benchmarking results. Now, switching to another framework does not help, as each benchmarking suite or workload generator has its own dependencies and significantly varies the configuration syntax and benchmark flow. *TexBench* abstracts all complexities into a single interface. The user now selects the candidate databases, picks or describes the workload, and launches all runs; results are automatically parsed and rendered side-by-side, with no cross-framework normalization or manual comparison. As a result, *TexBench* reduces the overall benchmarking time and enables users with no prior knowledge to initiate their first benchmark.

4 Related Work

Relational Benchmarking Suites. Benchmarking suites for relational data stores include several standard frameworks and benchmarks, such as TPC-C [47], TPC-DS [46], and TPC-E [48], constituting a well-developed field spanning across industry and research-focused workload generators. Recent work also introduces several application-specific benchmarks [16,30,34], including MUDD [44], Myriad [1], PDGF [35], that are curated to generate large-scale multi-table datasets. Each system targets its own specific requirements and challenges. For example, MUDD targets snowflake schemas for decision-support benchmarks by separating data distribution from data generation. However, these systems lack specialized controls needed for NoSQL storage engines.

Key-Value Benchmarks. Modern key-value workloads have become increasingly complex, changing across dimensions and exhibiting frequent diurnal and shifting patterns, where workload properties evolve over time. Cao *et al.* dissected three production workloads from Meta [7], including UDB, ZippyDB, and UP2X, highlighting the distinct characteristics of each. For example, UDB, a MySQL-compatible storage layer for the social graph, exhibits a diurnal write pattern that peaks around 17:00 PST and quickly drops to the bottom at 23:00 PST. These target workloads were emulated using YCSB and db_bench, where db_bench mimicked target workload patterns to some extent but still left room for improvement. The state-of-the-art key-value benchmarking suites and synthetic workload generators, such as YCSB, db_bench, and KVBench, do not capture these complex patterns.

YCSB. The Yahoo! Cloud Serving Benchmark (YCSB) [12] is a widely used benchmarking suite for cloud-native NoSQL databases, integrating 50+ databases

through a flexible design. YCSB supports various operations, including inserts, updates, point queries, range queries, and read-modify-write operations through its `CoreWorkload` module. However, it lacks support for point and range deletes altogether, and can only emulate static key-value workloads.

db_bench. db_bench [14] is a benchmarking tool offered by RocksDB [15], providing insights into its internal behavior and metrics. Like YCSB, db_bench also supports multiple operations, such as inserts, updates, point queries, range queries, point deletes, range deletes, and read-modify-writes. The tight coupling between db_bench and RocksDB makes it less useful as a general-purpose benchmarking suite, as it requires extensive engineering and development.

KVBench. KVBench [51] is a workload generator that supports a wide range of operations, including inserts, updates, empty and non-empty point queries, range queries and range deletes, and empty and non-empty point deletes. KVBench also supports operation-specific distributions and configurable – interleaved vs. sequential – ordering of operations. However, it does not support realistic workload patterns such as dynamically shifting and varied sortedness. It also lacks support for complex key-value formats and read-modify-writes.

Tectonic. The first key-value workload generator [33] that exploits the multi-dimensional space of workload generation, which includes several types of operations, operation-specific distributions, highly customized key construction, and fine-grained control over the entire workload. Given the rich design space, users are expected to write a lengthy, complex specification file to model their workload, a task that requires prior knowledge of the space. Tectonic also does not offer a unified, accessible interface for benchmarking databases against different standard benchmarks, and does not support visual comparison results.

5 Conclusion

In this work, we propose *TexBench*, a simplified, unified benchmarking suite for key-value data stores that captures modern application workloads with complex, dynamically shifting patterns. *TexBench* allows users to perform cross-benchmark evaluation using a single accessible user interface, eliminating the need to install and understand multiple benchmarking suites. Lastly, *TexBench* allows users to construct their own application-specific benchmarks tailored to their needs using the LLM-plugin and compare multiple databases' performance against them through visualizations. Currently, *TexBench* integrates four databases: ScyllaDB, RocksDB, Cassandra, and Redis, with several more in the integration pipeline, enabling users to benchmark key-value stores seamlessly. *TexBench* provides an extensible interface that industry developers can use to easily integrate and benchmark new databases.

Acknowledgements

We thank Artem Lavrov and Alexander H. Ott for contributing to the design and development of Tectonic.

References

1. Alexandrov, A., Schiefer, B., Poelman, J., Ewen, S., Bodner, T.O., Markl, V.: Myriad: parallel data generation on shared-nothing architectures. In: Proceedings of the 1st Workshop on Architectures and Systems for Big Data. pp. 30–33. ASBD '11, New York, NY, USA (2011). https://doi.org/10.1145/2377978.2377983, https://doi.org/10.1145/2377978.2377983
2. Apache: Cassandra. http://cassandra.apache.org (2023)
3. Apache: HBase. http://hbase.apache.org/ (2023)
4. Austin, J., Odena, A., Nye, M.I., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C.J., Terry, M., Le, Q.V., Sutton, C.: Program Synthesis with Large Language Models. CoRR **abs/2108.0** (2021), https://arxiv.org/abs/2108.07732
5. Balmau, O., Dinu, F., Zwaenepoel, W., Gupta, K., Chandhiramoorthi, R., Didona, D.: SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In: Proceedings of the USENIX Annual Technical Conference (ATC). pp. 753–766 (2019)
6. Borji, A.: A Categorical Archive of ChatGPT Failures. CoRR **abs/2302.0** (2023). https://doi.org/10.48550/ARXIV.2302.03494, https://doi.org/10.48550/arXiv.2302.03494
7. Cao, Z., Dong, S., Vemuri, S., Du, D.H.C.: Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In: Proceedings of the USENIX Conference on File and Storage Technologies (FAST). pp. 209–223 (2020)
8. Chandramouli, B., Prasaad, G., Kossmann, D., Levandoski, J.J., Hunter, J., Barnett, M.: FASTER: A Concurrent Key-Value Store with In-Place Updates. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 275–290 (2018), http://doi.acm.org/10.1145/3183713.3196898
9. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI). pp. 205–218 (2006), http://dl.acm.org/citation.cfm?id=1267308.1267323
10. Chatterjee, S., Jagadeesan, M., Qin, W., Idreos, S.: Cosine: A Cloud-Cost Optimized Self-Designing Key-Value Storage Engine. Proceedings of the VLDB Endowment **15**(1), 112–126 (2021)
11. Chatterjee, S., Pekala, M.F., Kruglyak, L., Idreos, S.: Limousine: Blending Learned and Classical Indexes to Self-Design Larger-than-Memory Cloud Storage Engines. Proceedings of the ACM on Management of Data (PACMMOD) **2**(1), 47:1—-47:28 (2024). https://doi.org/10.1145/3639302, https://doi.org/10.1145/3639302
12. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the ACM Symposium on Cloud Computing (SoCC). pp. 143–154 (2010), http://doi.acm.org/10.1145/1807128.1807152
13. Dong, S., Kryczka, A., Jin, Y., Stumm, M.: Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In: Proceedings of the USENIX Conference on File and Storage Technologies (FAST). pp. 33–49 (2021), https://www.usenix.org/conference/fast21/presentation/dong
14. Facebook: db_bench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools (2024)
15. Facebook: RocksDB. https://github.com/facebook/rocksdb (2024)

16. Frank, M., Poess, M., Rabl, T.: Efficient update data generation for DBMS benchmarks. In: Third Joint WOSP/SIPEW International Conference on Performance Engineering, ICPE'12, Boston, MA, USA - April 22 - 25, 2012. pp. 169–180 (2012), https://doi.org/10.1145/2188286.2188315
17. Google: LevelDB. https://github.com/google/leveldb/ (2021)
18. Huynh, A., Chaudhari, H.A., Terzi, E., Athanassoulis, M.: Towards flexibility and robustness of LSM trees. The VLDB Journal pp. 1–24 (2024), https://doi.org/10.1007/s00778-023-00826-9
19. Idreos, S., Callaghan, M.: Key-Value Storage Engines. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 2667–2672 (2020)
20. Im, J., Bae, J., Chung, C., Arvind, Lee, S.: PinK: High-speed In-storage Key-value Store with Bounded Tails. In: Proceedings of the USENIX Annual Technical Conference (ATC). pp. 173–187 (2020), https://www.usenix.org/conference/atc20/presentation/im
21. Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y., Madotto, A., Fung, P.: Survey of Hallucination in Natural Language Generation. ACM Computing Surveys **55**(12), 248:1—-248:38 (2023). https://doi.org/10.1145/3571730, https://doi.org/10.1145/3571730
22. Jin, Y., Tseng, H.W., Papakonstantinou, Y., Swanson, S.: KAML: A Flexible, High-Performance Key-Value SSD. In: Proceedgins of the IEEE International Symposium on High Performance Computer Architecture (HPCA). pp. 373–384 (2017), https://doi.org/10.1109/HPCA.2017.15
23. Kanellis, K., Chandramouli, B., Hart, T., Venkataraman, S.: From FASTER to F2: Evolving Concurrent Key-Value Store Designs for Large Skewed Workloads. Proceedings of the VLDB Endowment **18**(12), 4910–4923 (2025). https://doi.org/10.14778/3750601.3750615, https://www.vldb.org/pvldb/vol18/p4910-kanellis.pdf
24. Kanellis, K., Chandramouli, B., Venkataraman, S.: F2: Designing a Key-Value Store for Large Skewed Workloads. CoRR **abs/2305.0** (2023), https://doi.org/10.48550/arXiv.2305.01516
25. Kaushik, S., Athanassoulis, M., Sarkar, S.: RangeReduce: Query-Driven LSM Compactions. In: Proceedings of the IEEE International Conference on Data Engineering (ICDE) (2026)
26. Kaushik, S., Sarkar, S.: Anatomy of the LSM Memory Buffer: Insights & Implications. In: Proccedings of the International Workshop on Testing Database Systems (DBTest). pp. 23–29 (2024), https://doi.org/10.1145/3662165.3662766
27. Keahey, K., Anderson, J., Zhen, Z., Riteau, P., Ruth, P., Stanzione, D., Cevik, M., Colleran, J., Gunawi, H.S., Hammock, C., Mambretti, J., Barnes, A., Halbach, F., Rocha, A., Stubbs, J.: Lessons Learned from the Chameleon Testbed. In: Proceedings of the USENIX Annual Technical Conference (ATC). pp. 219–233 (2020), https://www.usenix.org/conference/atc20/presentation/keahey
28. Lao, J., Wang, Y., Li, Y., Wang, J., Zhang, Y., Cheng, Z., Chen, W., Tang, M., Wang, J.: GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization. Proceedings of the VLDB Endowment **17**(8), 1939–1952 (2024). https://doi.org/10.14778/3659437.3659449, https://www.vldb.org/pvldb/vol17/p1939-tang.pdf
29. Li, Y., Liu, Z., Lee, P.P.C., Wu, J., Xu, Y., Wu, Y., Tang, L., Liu, Q., Cui, Q.: Differentiated Key-Value Storage Management for Balanced I/O Performance. In: Proceedings of the USENIX Annual Technical Conference (ATC). pp. 673–687 (2021), https://www.usenix.org/conference/atc21/presentation/li-yongkun

30. Li, Y., Zhang, R., Li, Y., Ke, S., Zhang, S., Zhou, A.: Lauca: Generating Application-Oriented Synthetic Workloads. CoRR **abs/1912.0** (2019), http://arxiv.org/abs/1912.07172
31. Liu, N.F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., Liang, P.: Lost in the Middle: How Language Models Use Long Contexts. Trans. Assoc. Comput. Linguistics **12**, 157–173 (2024). https://doi.org/10.1162/TACL_A_00638, https://doi.org/10.1162/tacl{_}a{_}00638
32. OpenAI: GPT-4 Technical Report. CoRR **abs/2303.0** (2023). https://doi.org/10.48550/ARXIV.2303.08774, https://doi.org/10.48550/arXiv.2303.08774
33. Ott, A.H., Kaushik, S., Chen, B., Sarkar, S.: Tectonic: Bridging Synthetic and Real-World Workloads for Key-Value Benchmarking. In: Performance Evaluation and Benchmarking - TPC Technology Conference (TPCTC). vol. 16261, pp. 69–88 (2025). https://doi.org/10.1007/978-3-032-18070-4_5, https://doi.org/10.1007/978-3-032-18070-4{%}5C{_}5
34. Rabl, T., Lang, A., Hackl, T., Sick, B., Kosch, H.: Generating Shifting Workloads to Benchmark Adaptability in Relational Database Systems. In: Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers. pp. 116–131 (2009), https://doi.org/10.1007/978-3-642-10424-4{_}9
35. Rabl, T., Poess, M.: Parallel data generation for performance analysis of large, complex RDBMS. In: Proceedings of the International Workshop on Testing Database Systems (DBTest). p. 5 (2011), https://doi.org/10.1145/1988842.1988847
36. Redis: Online reference. http://redis.io/
37. Rockset: Rockset. https://rockset.com (2024)
38. Sarkar, S., Athanassoulis, M.: Dissecting, Designing, and Optimizing LSM-based Data Stores. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 2489–2497 (2022), https://www.youtube.com/watch?v=hkMkBZn2mGs
39. Sarkar, S., Chen, K., Zhu, Z., Athanassoulis, M.: Compactionary: A Dictionary for LSM Compactions. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 2429–2432 (2022)
40. Sarkar, S., Dayan, N., Athanassoulis, M.: The LSM Design Space and its Read Optimizations. In: Proceedings of the IEEE International Conference on Data Engineering (ICDE). pp. 3578–3684 (2023), https://doi.org/10.1109/ICDE55515.2023.00273
41. Sarkar, S., Papon, T.I., Staratzis, D., Zhu, Z., Athanassoulis, M.: Enabling Timely and Persistent Deletion in LSM-Engines. ACM Transactions on Database Systems (TODS) **48**(3), 8:1—-8:40 (2023), https://doi.org/10.1145/3599724
42. Sarkar, S., Staratzis, D., Zhu, Z., Athanassoulis, M.: Constructing and Analyzing the LSM Compaction Design Space. Proceedings of the VLDB Endowment **14**(11), 2216–2229 (2021), http://vldb.org/pvldb/vol14/p2216-sarkar.pdf
43. ScyllaDB: Online reference (2024), https://www.scylladb.com/
44. Stephens, J.M., Poess, M.: MUDD: a multi-dimensional data generator. In: Proceedings of the International Workshop on Software and Performance (WOSP). pp. 104–109 (2004), https://doi.org/10.1145/974044.974060
45. Team, L.: The Llama 3 Herd of Models. CoRR **abs/2407.2** (2024). https://doi.org/10.48550/ARXIV.2407.21783, https://doi.org/10.48550/arXiv.2407.21783
46. TPC: Specification of the TPC-DS benchmark. http://www.tpc.org/tpcds/
47. TPC: Specification of TPC-C benchmark. http://www.tpc.org/tpcc/
48. TPC: Specification of TPC-E benchmark. http://www.tpc.org/tpce/

49. Xue, S., Qi, D., Jiang, C., Cheng, F., Chen, K., Zhang, Z., Zhang, H., Wei, G., Zhao, W., Zhou, F., Yi, H., Liu, S., Yang, H., Chen, F.: Demonstration of DB-GPT: Next Generation Data Interaction System Empowered by Large Language Models. Proceedings of the VLDB Endowment **17**(12), 4365–4368 (2024). https://doi.org/10.14778/3685800.3685876, https://www.vldb.org/pvldb/vol17/p4365-chen.pdf
50. Zhao, C., Chugh, T., Min, J., Liu, M., Krishnamurthy, A.: Dremel: Adaptive Configuration Tuning of RocksDB KV-Store. Proc. ACM Meas. Anal. Comput. Syst. **6**(2), 37:1—-37:30 (2022). https://doi.org/10.1145/3530903, https://doi.org/10.1145/3530903
51. Zhu, Z., Saha, A., Athanassoulis, M., Sarkar, S.: KVBench: A Key-Value Benchmarking Suite. In: International Workshop on Testing Database Systems (DBTest). pp. 9–15 (2024), https://doi.org/10.1145/3662165.3662765
52. Zhu, Z., Sarkar, S., Athanassoulis, M.: Acheron: Persisting Tombstones in LSM Engines. In: Companion of the International Conference on Management of Data (SIGMOD). pp. 131–134 (2023)